International Journal of Software Engineering
and Knowledge Engineering
Vol. 19, No. 6 (2009) 765–789

World Scientific
www.worldscientific.com

AN EFFICIENT LOTOS-BASED FRAMEWORK FOR DESCRIBING AND SOLVING (TEMPORAL) CSPs

SAMIRA SADAOUI*, MALEK MOUHOUB† and BO CHEN
Computer Science Department, University of Regina,
Regina SK, Canada S4S 0A2
**sadaouis@cs.uregina.ca*
†*mouhoubm@cs.uregina.ca*

Received 29 March 2006
Revised 1 October 2007
Accepted 27 June 2008

Simulation of complex Lotos specifications is not always efficient due to the space explosion problem of their corresponding transition systems. To overcome this difficulty in practice, we present in this paper a novel approach which integrates constraint propagation techniques into the Lotos specifications. These solving techniques are used to reduce the size of the search space before and during the search for a solution to a given combinatorial problem under constraints. In order to do that, we first tackle the challenging task of describing combinatorial problems in Lotos using the Constraint Satisfaction Problem (CSP) framework. In this regard, we provide two generic Lotos templates for describing CSPs and temporal CSPs (CSPs involving temporal constraints). To evaluate the time performance of the framework we propose, we have conducted several experimental tests on instances of the N-Queens, the machine scheduling and randomly generated CSPs. The results of these experiments are promising and demonstrate the efficiency of Lotos simulation when CSP techniques are integrated.

Keywords: Formal specifications; lotos; simulation; constraint satisfaction; temporal reasoning.

1. Introduction

Our goal in this paper is to use constraint propagation techniques in order to improve the simulation phase running time of Lotos specifications. This will enable us to simulate complex specified combinatorial problems in very efficient running time. In order to do that, we first tackle the challenging task of describing combinatorial problems in Lotos using the Constraint Satisfaction Problem (CSP) framework. In this regard, we provide in this paper two generic Lotos templates to facilitate the transformation of a combinatorial problem into a (temporal) CSP.[a] The resulting Lotos specifications are then solved by incorporating existing constraint

[a]A temporal CSP is a CSP involving temporal constraints.

propagation techniques. To our best knowledge, this is the first time such approach for solving combinatorial problems has been adopted.

A CSP is composed of a list of variables defined on finite domains of values and a list of relations restricting the values that the variables can simultaneously take [1–4]. A solution to a CSP is a set of assigned values to variables that satisfy all the constraints. CSPs are very powerful for representing discrete combinatorial problems including frequency assignment, configuration and conceptual design, molecular biology, chemical hypothetical reasoning, and scene analysis. Since a CSP is known to be an NP-Hard problem in general,[b] a backtrack search algorithm of exponential time cost is needed to find a complete solution. To overcome this difficulty, constraint propagation techniques have been proposed [1, 3–5]. These techniques consist of reducing the size of the search space before and during the backtrack search. This is achieved by removing from the domains of the variables some values that do not belong to any solution.

Lotos is the ISO[c] formal specification language for describing, simulating and verifying concurrent and distributed systems [6]. Lotos is composed of two complementary parts: (i) a process algebra that expresses temporal relations of system actions, and (ii) algebraic data types that represent data structures and value expressions. Lotos provides the ability to describe complex data structures by composition and extension mechanisms. And the Lotos equations can concisely specify complex constraints. Many tools have been developed for Lotos. The most powerful one is CADP (CAESAR/ALDEBARAN Development Package) toolbox [7]. Indeed, CADP is a complete design tool for the compilation, simulation (execution), verification (with model-checking), testing and rapid prototyping of Lotos systems. The compilation phase generates a labelled transition system (LTS) which encodes all the possible execution sequences of a specification. To efficiently handle types, variables and operations, CADP applies concrete implementation by translating data types into libraries of C types and functions.

In practice the simulation of complex systems is not always efficient due to the space explosion problem of LTS [8]. Many methods [8–11] have been proposed to cope with this difficulty. In this paper, we address this problem in a new way: first by expressing CSPs in Lotos and then by using efficient CSP solving techniques to improve the simulation (execution) time cost. As mentioned earlier, the choice for CSPs is motivated by the fact that this framework is very powerful for solving discrete combinatorial problems as indicated in [2, 12]. This is the reason why the most popular constraint solvers such as ILOG SOLVER [13] and Prolog [14] include a library of constraint propagation techniques.

In this paper, the challenge is how to integrate CSP techniques into Lotos specifications in order to provide a flexible and efficient way for solving large complex

[b]There are special cases where CSPs are solved in polynomial time, for instance, the case where a CSP network is a tree [1, 4].
[c]International Organization for Standardization.

systems such as combinatorial problems. Therefore we define a Lotos-based (temporal) CSP framework composed of the following.

- A library of data-types whose operations and equations are expressive enough to clearly and concisely specify complex constraints. In addition to facilitate the description of constraint applications, we provide two generic templates that can be customized to represent any CSP and temporal CSP.
- The constraint propagation algorithms [1–4] used to significantly improve the performance of problem solving. These algorithms (that we have implemented here in C) are integrated into Lotos specifications through the *external implementation* of Lotos data types.
- The CADP simulators to find solutions to the Lotos CSP specifications.

Our approach makes a clear separation between CSP specifications and CSP algorithms in order to freely apply any combination of CSP algorithms and also to include new CSP techniques in the future. In the particular case of temporal CSPs (CSPs involving metric and symbolic temporal constraints), we integrate the TemPro modeling framework [15,16] into the Lotos specifications. This will enable us to simulate a wide variety of real-life applications such as scheduling, planning, temporal databases, molecular biology and any other applications under temporal constraints. The choice for TemPro is motivated by the fact that this latter framework is heavily based on the CSP framework that we have adopted for general constraint problems.

The rest of the paper is organized as follows. Sections 2 and 3 discuss CSPs and temporal CSPs respectively. Section 4 presents two generic templates for describing CSPs and temporal CSPs. It also shows how to instantiate these templates for specific constraint problems. Section 5 describes how to integrate constraint propagation and TemPro into Lotos specifications. Section 6 describes the experimental tests conducted on instances of the N-Queens, the machine scheduling and randomly generated CSPs, in order to evaluate the efficiency of our (temporal) CSP framework. Concluding remarks and some perspectives are finally covered in Sec. 7.

2. Constraint Propagation for CSPs

In order to describe CSPs and the constraint propagation techniques for solving them, let us consider the following example of the N-Queens problem.

Example 1 (N-Queens). Given any integer N, the problem is to place N queens on N distinct squares (each queen has to be placed on a different row or a different column) in an N×N chessboard satisfying the constraints that no two queens should threaten each other, that is no two queens can be placed on the same row, same column, and same diagonal.

Figure 1 represents the example of the 4-Queens and its CSP representation using a graph. Indeed a binary CSP (CSP where the constraints are binary relations)

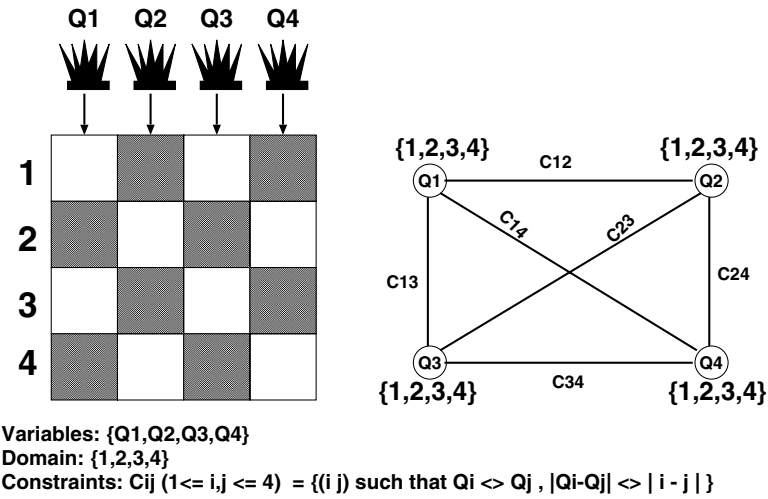

Fig. 1. The 4-Queens problem and its CSP representation using a graph.

is usually represented by a graph where nodes correspond to variables and arcs represent the binary constraints between variables. In this case, each of the 4 queens has to be placed on a different column.

The basic way to solve a CSP is the systematic search called Backtracking (BT) which explores the search space looking for a possible solution satisfying all the constraints. BT incrementally attempts to extend a partial solution towards a complete one, by repeatedly choosing a value for another variable [1, 12]. Figure 2 traces the BT algorithm for solving the 4-Queens. Here the search space, represented by the

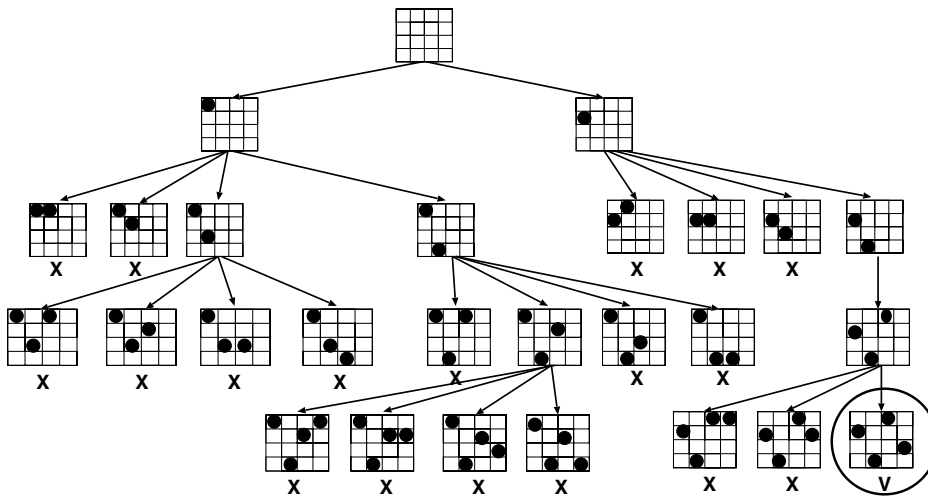

Fig. 2. Backtracking algorithm applied to the 4-Queens.

tree of possibilities, is explored from left to right. The first queen is first placed in the first row, then we look for a position for the second queen such that there is no conflict with the first one. Once this is obtained, we look for a position for the third queen that does not conflict with the positions of the first two queens. As we can see from Fig. 2, after several attempts we realize that it is not possible to place the third queen because of the position of the first one. The first queen has then to be moved to the second position as illustrated by the right subtree. This late detection of the inconsistent value for the first queen is the main disadvantage of the BT algorithm. In order to overcome this problem, constraint propagation techniques have been proposed [1–3,12]. The goal of these techniques is to prune earlier late failure. For example in the case of the 4-Queens, it would be better to know beforehand that placing the first queen in position one will not lead to a solution. This will indeed save the many tries depicted by the left tree of Fig. 2. The way constraint propagation works is by propagating the impact of a given decision (assignment of a value to a variable) to the domains of the variables that are not yet assigned. For instance, placing the first queen in position one means that there is no way to place the second queen in position one and two, the third queen in position one and three and the fourth queen in position one and four. These positions for queens 2, 3 and 4 have then to be removed from their respective domain. This kind of elimination is handled by the Arc Consistency (AC) algorithm [3,12] which constitutes the core of constraint propagation. AC consists of enforcing a local consistency between each pair of variables of the CSP. More precisely, for each pair of variables (X_i, X_j) AC removes any value a from X_i's domain that has no support in X_j's domain (no value b in X_j's domain such that the pair (a, b) satisfies the relation between X_i and X_j). For instance, in the example of the 4-Queens, when applying AC after placing queen 1 in position 1, positions 1 and 3 for queen 2 should be removed (since these two positions conflict with position 1 of queen 1). There are two main strategies for applying AC in the constraint propagation process [1,12]. The first one is called Forward Checking (FC)[d] and consists of enforcing AC between the currently assigned variable and the future ones (variables that are not yet assigned). Figure 3 illustrates FC applied to the 4-Queens. As we can easily see with FC the search space is considerably reduced (compared to BT). The second strategy, called Full Look Ahead (FLA), does more consistency check than FC. Indeed, in addition to enforcing AC between the current and the future variables (case of FC), FLA applies AC between every pair of the future variables. Figure 4 describes FLA applied to the 4-Queens. Note that while the search space corresponding to FLA is smaller than the one of FC, there is more time effort spent after each variable assignment in the case of FLA. A comparison, in terms of running time needed to obtain a solution, between FC and FLA has been carried out in [12] and favors FC in most of the cases.

[d]This strategy is also called partial look ahead [1].

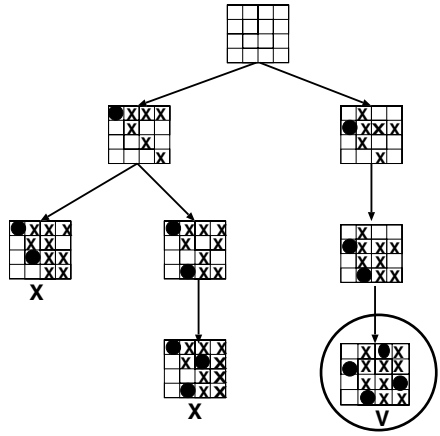

Fig. 3. Forward Checking applied to the 4-Queens.

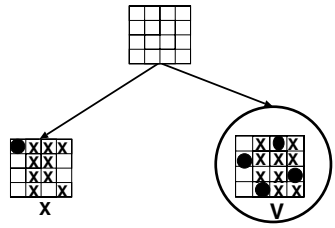

Fig. 4. Full Look Ahead applied to the 4-Queens.

Note that, in addition to AC, other forms of local consistency have been proposed [3, 5]. For example node consistency which consists of checking the consistency according to unary constraint of each variable, path consistency which checks the consistency between any subset of three variables and in the general case, k-consistency which checks the consistency between any subset of k variables. AC is still however the most popular and the most efficient form (in terms of running time for finding a solution) of local consistency [12].

The most popular AC algorithm has been developed by Mackworth [3] and is called AC-3.[e] While AC-3 has been proposed over 20 years ago, it remains one of the easiest to implement and understand today. Figure 5 illustrates the code of AC-3. AC-3 uses the list Q of the variable pairs to be processed. This list is initialized with all pairs of variables sharing a constraint. The algorithm will then pick a pair of variables from Q and applies the 2 consistency on it through the function *REVISE*. More precisely, when applied to the variable pair (i, j) the function *REVISE* will remove any value a, from the i's domain, that has no support in j's domain. For

[e]AC-3 is an improvement of AC-1 and AC-2, both proposed by the same author [3].

Function *REVISE*(i, j)
1. *REVISE* $\leftarrow$ *false*
2. **For** each value $a \in Domain_i$ **Do**
3. **If** there is no $b \in Domain_j$ such that *compatible*(a, b) **Then**
4. remove a from $Domain_i$
5. *REVISE* $\leftarrow$ *true*
6. **End-If**
7. **End-For**

Algorithm AC-3
1. Given a CSP = (X,D,C)
   (*X: set of variables, D: set of the variable domains, C: set of constraints*)
2. $Q \leftarrow \{(i, j) \mid i, j \in X$ and $(i, j) \in C\}$ (list initialized to all pairs sharing a constraint)
3. **While** $Q \neq Nil$ **Do**
4. $Q \leftarrow Q - \{(i, j)\}$
5. **If** *REVISE*(i, j) **Then**
6. $Q \leftarrow Q \sqcup \{(k, i) \mid (k, i) \in C \wedge k \neq j\}$
7. **End-If**
8. **End-While**

Fig. 5. Pseudo code of the algorithm AC-3.

that, the function *compatible* checks if there is a compatibility (according to the constraint between i and j) between a and a value b of j's domain.

There have been many attempts to best the worst case time complexity of AC-3[f] and though in theory these other algorithms (namely AC-4 [17], AC-5 [18], AC-6 [19] and AC-7 [20]) have better worst case time complexities, they are harder to implement. Indeed, while AC-3 only uses one data structure containing the list of pairs to be checked, AC-4, AC-6 and AC-7 are based on very complex data structures that maintain the support of each variable value during the arc consistency process. On the other hand AC-5 deals with special classes of relations, such as monotonic and functional constraints. Also the AC-4 algorithm fares worse on average time complexity than the AC-3 algorithm [21]. It was not only until recently when Zhang and Yap [22][g] proposed an improvement directly derived from the AC-3 algorithm into their algorithm AC-3.1. The worst case time complexity of AC-3 is bounded by $O(ed^3)$ [4] where e is the number of constraints and d is the domain size of the variables (number of values with the variable domain). In fact this complexity depends mainly on the way the function *REVISE*, in Fig. 5 above, is enforced for each variable pair. Indeed, if anytime a given pair (i, j) is processed, a support for each value from the domain of i is searched from scratch in the domain of j, then the worst case time complexity of AC-3 is $O(ed^3)$. Instead of a search from scratch, Zhang and Yap [22] proposed a new view that allows the search to resume from the point where it stopped in the previous revision of (i, j). By doing so, the worst

[f]The worst case time complexity of AC-3 is majored by $O(ed^3)$ where e is the number of constraints and d is the domain size (number of values within the domain) of the variables.
[g]Another arc consistency algorithm (called AC-2001) based on the same idea as AC-3.1 was proposed by Bessière and Régin [23]. We have chosen AC-3.1 for the simplicity of its implementation.

Function $EXISTb((i,a),j)$
1. $b \leftarrow ResumePoint((i,a),j)$
($ResumePoint((i,a),j)$ *remembers the first value* b *such that* $compatible(a,b)$ *is true in the previous revision of* (i,j))
2. **If** $b \in Domain_j$ **Then**
3. return true
4. **Else**
5. **While** $b \leftarrow successor(b, Domain_j^0)$ and $b \neq NIL$
($Domain_j^0$ *denotes the domain of* j *before arc consistency*)
($successor(b, Domain_j^0)$ *returns the successor of* b *in* $Domain_j^0$)
6. **If** $b \in Domain_j$ and $compatible(a,b)$ **Then**
7. $ResumePoint((i,a),j) \leftarrow b$
8. return true
9. **End-If**
10. **End-While**
10. return false
11. **End-If**

Fig. 6. Pseudo code of AC-3.1: function for searching b in line 3 of $REVISE(i,j)$.

case time complexity of AC-3 is achieved in $O(ed^2)$. This new idea is implemented by the function $EXISTb$ in Fig. 6. This function has to be called in line 3 of the function $REVISE$, Fig. 5 as follows.

3. If $EXISTb((i,a),j)$ **Then**

3. Constraint Propagation for Temporal CSPs

In the past years [16,24] we have proposed a modeling framework, based on CSPs, for managing the particular case of temporal constraints. This framework, that we call TemPro, translates a given problem under numeric and symbolic time constraints into a particular CSP that we call Temporal CSP (or TCSP). In the same way as for CSPs, we use constraint propagation techniques (that we have modified in order to handle time constraints [16]) to solve the TCSP in an efficient manner.

More precisely a TCSP is a CSP where:

- The variables are called **temporal events**. A temporal event is a temporal information that holds (is true) over a numeric time interval. For instance: *John is reading the paper for 10 minutes* is an event that lasts 10 minutes.
- The domain of an event is a finite and discrete set of intervals, with constant duration. We call this domain the Set Of Possible Occurrences (or SOPO) that the event can take. Figure 8 illustrates a SOPO (domain) of a given event. As we can see in the figure, the SOPO is defined by the parameters: `Begintime` and `Endtime` representing the earliest start and latest end times of the event; `Duration`, representing the duration of the event and `Step`, denoting the discretisation step used (number of time units between the starting time of two adjacent numeric intervals within the SOPO). Indeed, we use here a discrete representation of time. To illustrate the SOPO of an event, let us consider the example we provided in

the item above with the following modification: *John is reading the paper for 10 minutes between 7am and 7:20am.* If the time unit used here is the minute then we will have the following SOPO: [0,20,10,1] where 0, 20, 10 and 1 are respectively the `Begintime, Endtime, Duration` and `Step` of the SOPO (given that 7am is chosen as the time origin).

- A Constraint is represented by the qualitative relation between a pair of events. This relation is denoted by a disjunction of Allen primitives [25]. Indeed, Allen has defined 13 possible primitives that can hold between a pair of numeric intervals (see Fig. 7 for the definition of the 13 *Allen* primitives) and a disjunction of some of these primitives can be used to express the relative position between events. For instance, the following sentence:
John is reading the paper while he is drinking a cup of tea
can be represented by the relation:

$$m \vee mi \vee o \vee oi \vee d \vee di \vee s \vee si \vee f \vee fi \vee eq$$

between the events *reading the paper* and *drinking a cup of tea.*

The following example illustrates the transformation of a problem including numeric and symbolic temporal constraints into a TCSP using the model TemPro.

Relation	Symbol	Inverse	Meaning
X before Y	b	bi	
X meets Y	m	mi	
X overlaps Y	o	oi	
X during Y	d	di	
X starts Y	s	si	
X finishes Y	f	fi	
X equals Y	eq	eq	

X: Y:

Fig. 7. Allen primitives.

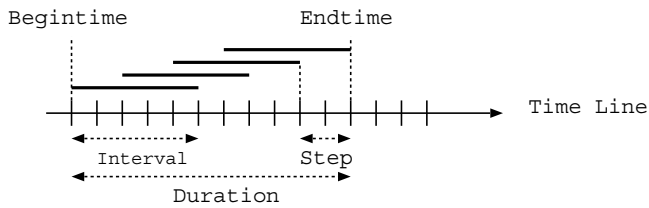

Fig. 8. A SOPO (domain) of a given event.

Example 2 (Machine Scheduling Problem). The production of five items A, B, C, D and E requires three mono processor machines M_1, M_2 and M_3. Each item can be produced using two different ways depending on the order in which the machines are used. The process time of each machine is variable and depends on the task to be processed. The following lists the different ways to produce each of the five items (the process time for each machine is mentioned in brackets):

item A: $M_2(3), M_1(3), M_3(6)$ or
$M_2(3), M_3(6), M_1(3)$
item B: $M_2(2), M_1(5), M_2(2), M_3(7)$ or
$M_2(2), M_3(7), M_2(2), M_1(5)$
item C: $M_1(7), M_3(5), M_2(3)$ or
$M_3(5), M_1(7), M_2(3)$
item D: $M_2(4), M_3(6), M_1(7), M_2(4)$ or
$M_2(4), M_3(6), M_2(4), M_1(7)$
item E: $M_2(6), M_3(2)$ or
$M_3(2), M_2(6)$

The goal here is to find a possible schedule of the different machines to produce the five items and respecting all the constraints of the problem. In the following, we will describe how is the above problem transformed into a TCSP using our model TemPro. Figure 9 illustrates the graph representation of the sub TCSP needed to produce items A and B. Each node of the graph represents a given event. Arcs

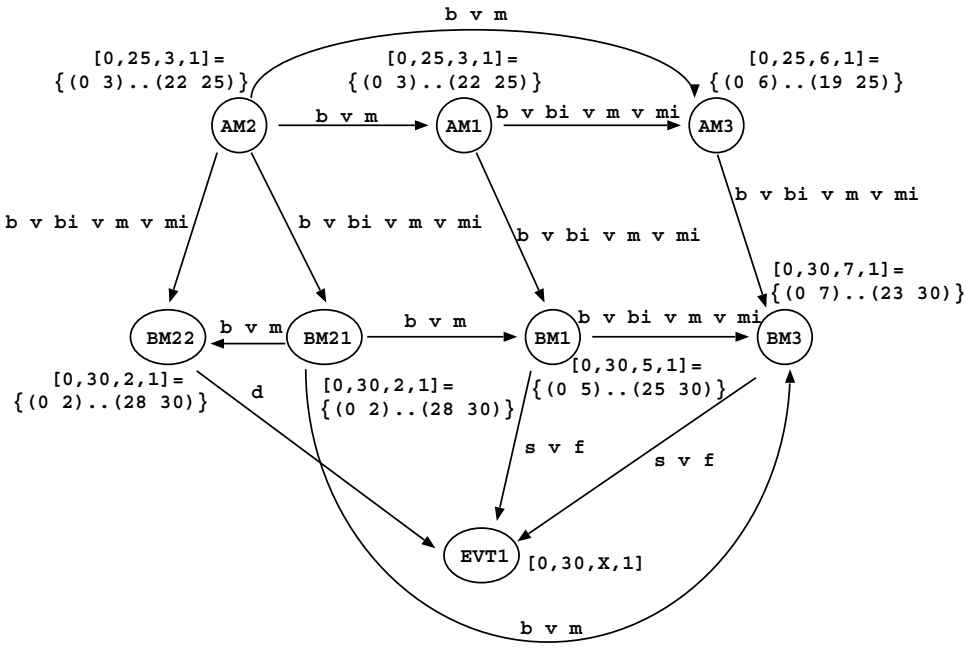

Fig. 9. TCSP corresponding to a subset of the problem presented in Example 2.

represent the constraints between events and are libelled by the corresponding relations. We assume that items A and B should be produced within 25 and 30 units of time respectively. A temporal event corresponds here to the contribution of a given machine to produce a certain item. For example, AM_1 corresponds to the use of machine M_1 to produce the item $A, \ldots$, etc. In the particular case of item B, machine M_2 is used twice. Thus, there are two corresponding events: BM_{21} and BM_{22}. 16 events are needed in total to produce the five items. Most of the qualitative information can easily be represented by the disjunction of Allen primitives. For example, the constraint (disjunction of two sequences) needed to produce item A is represented by the following three relations:

$AM_2 \;\; b \vee m \;\; AM_1$
$AM_2 \;\; b \vee m \;\; AM_3$
$AM_1 \;\; b \vee m \vee bi \vee mi \;\; AM_3$

We present in the following the resolution method based on constraint propagation for solving TCSPs. The method involves two main stages. A filtering stage in which arc consistency techniques and a numeric → symbolic conversion method are used to reduce the size of the search space by removing some inconsistent values. The backtrack search phase is then used to look for a possible solution. More precisely, our solving method works as follows.

Numeric → Symbolic Conversion. Perform the numeric → symbolic conversion on all the constraints. If one symbolic relation becomes empty then the constraint network is not consistent. The numeric → symbolic conversion works as follows: from the numeric information, we can extract the corresponding symbolic relation. An intersection of this relation with the given qualitative information will reduce the size of the latter which simplifies the size of the original problem.

Arc Consistency. Perform the arc consistency algorithm AC-3 [22, 23] on the temporal windows. If the new graph is not arc consistent then it is not consistent.

Backtrack Search. Perform a backtrack search algorithm in order to look for a possible solution to the problem. The arc consistency algorithm is used here during the backtrack search in order to prevent earlier later failure.

Figure 10 illustrates the solution to the problem of Example 1, provided by the above solving method. Note that this solution is optimal[h] but not unique.

4. Lotos Specification of CSPs and TCSPs

In [26, 27], we have proposed a first approach to describe and solve CSPs with Lotos. The sorts and operations of Lotos represent the domains, variables and constraints of CSPs. The Lotos equations define the constraints between CSP variables. Using Lotos and CADP toolbox, a CSP can be solved in several ways: checking the

[h] The total processing time of all machines needed to produce the five items, 26 seconds, is minimal.

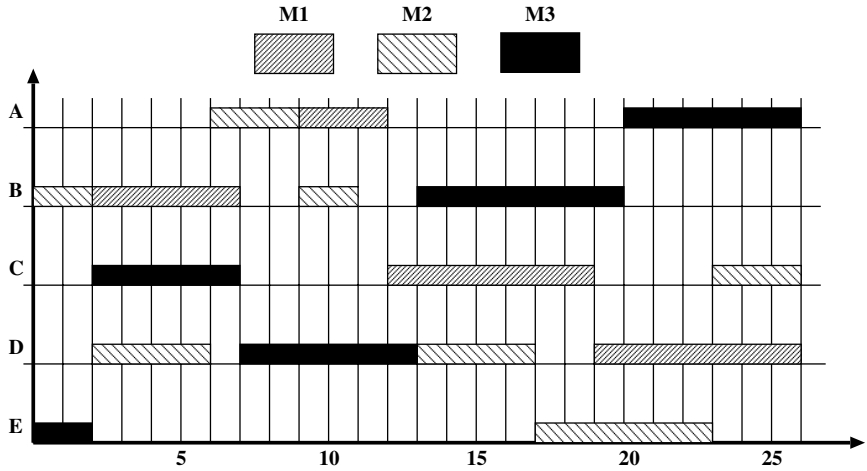

Fig. 10. Optimal solution provided by the constraint propagation based method.

consistency, finding all the possible solutions, checking if a path is a solution or completing a partial solution. This approach is however not general enough to specify complex CSPs. In addition, the problem solving (or simulation) is really time consuming. For instance, it takes almost eight seconds to find a solution for the 4-Queens problem. Consequently, we build in Lotos a reusable data-type library which is expressive enough to represent any CSP and TCSP as well. The most important types are given in Table 1. In order to facilitate the specification of any constraint applications, we extend this library with two generic Lotos templates for describing CSPs and TCSPs.

4.1. *A Lotos template for CSPs*

In the following, we first introduce the CSP template and then we show how to customize it through two examples. As shown in Fig. 11, the Lotos template is

Table 1. A library for specifying CSPs and TCSPs.

Data Type in Lotos	Definition
Variable	CSP variable
VariableSet	Set of variables
Domain	Domain of a variable
VariableDomainSet	Set of domains of variables
Value	Possible values of a variable
Constraint	Set of value pairs (possible assignments of two variables)
ConstraintSet	Set of constraints
SOPO	Specification of SOPO
Allen	Specification of thirteen Allen primitives
ConstraintProblem	Set of variables, domains and constraints

```
type CSPTemplate is (*Uses*) ConstraintProblem
  sorts FixVar (*Set of domains*)
  opns (*Operations*)
       (*Enter CSP Variables*)
       varNum: -> Nat (*Number of variables*)
       dSize:FixVar -> Nat (*Domain size*)
       id(*!external implementation*):FixVar -> Nat
       rel:FixVar,FixVar -> Bool (*Relation*)
       sat:FixVar,Nat,FixVar,Nat -> Bool (*Satisfaction*)
       \_eq\_,\_ne\_:FixVar,FixVar -> Bool (*Default operations*)
  eqns (*Equations*)
       forall x,y:FixVar, m,n:Nat
       ofsort Nat
        (*Enter number of variables with varNum*)
        (*Enter domain size of each variable with dSize()*)
       ofsort Bool
        (*Define relations between two variables with rel()*)
        (*Define satisfaction with sat()*)
        (*Default equations*)
        x eq x = true; x eq y = false; x ne y = not (x eq y);
endtype
```

Fig. 11. A template for CSPs.

composed of a set of operations and a set of equations. Each operation must be defined with equations.

The Lotos description of a specific CSP consists of instantiating the CSP template as shown below.

- Enter the names of CSP variables. Variables are enumerable, and each one is given a meaningful name. We can also use natural numbers to represent the variables when they are many.
- Enter the number of variables through the constant *varNum*.
- Enter the domain size of each variable through the operation *dSize*(). Variables may have different domains. For example, one variable may have a domain of strings while another one has a domain of natural numbers. If variables have the same domain, they may have different set of values.
- Enter relations and satisfaction. The constraints between variables are described via the two operations *rel*() and *sat*(). *rel*() checks whether there is a relation between two variables and *sat*() defines the CSP constraints.

We now show how to apply the CSP template on the 3-map coloring problem given below.

Example 3 (Map Coloring). Given a graph with six vertexes and eight edges, we want to assign one color from a set of three colors (red, blue, green) to each vertex such that no two adjacent vertexes have the same color.

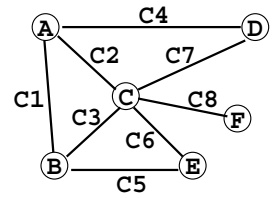

Fig. 12. Graph representation of the 3-map coloring.

As illustrated in Fig. 12, the CSP representation of this problem can be described with six variables, A to F, and eight constraints, C_1 to C_8, where C_1 means "A <> B", C_2 "A <> C", and so on. Here all the variables have the same domain: {red, blue, green}. Note that in Fig. 12 we use the graph representation of a CSP where nodes correspond to variables and arcs represent the binary constraints between variables.

In Fig. 13, we instantiate the CSP template to generate the specification of the map coloring. During the instantiation process, the user only needs to enter the CSP variables, their domain size and constraints. In this example, *rel*() denotes which two vertexes are adjacent. Here all the variables have the same constraint *sat*() which defines that "any two adjacent vertexes cannot have the same color". The simulation of the 3-map coloring specification generates twelve solutions. For instance, the

```
type 3-MapColoring is ConstraintProblem
  sorts FixVar
  opns (*Enter CSP Variables*)
       A, B, C, D, E, F: -> FixVar
       varNum: -> Nat
       dSize:FixVar -> Nat
       id(*!external implementation*):FixVar -> Nat
       rel:FixVar,FixVar -> Bool
       sat:FixVar,Nat,FixVar,Nat -> Bool
       \_eq\_,\_ne\_:FixVar,FixVar -> Bool
  eqns forall x,y:FixVar, m,n:Nat
       ofsort Nat
        (*Enter number of variables with varNum*)
         varNum = 6;
        (*Enter domain size of each variable with dSize()*)
         dSize(x) = 3; (*All variables have the same domain*)
       ofsort Bool
        (*Define relations between two variables with rel()*)
         rel(A,B) = true; rel(A,D) = true; rel(A,C) = true;
         rel(C,B) = true; rel(C,D) = true; rel(C,E) = true; rel(C,F) = true;
        (*Define satisfaction with sat()*)
         sat(x,m,y,n) = m ne n;
         x eq x = true; x eq y = false; x ne y = not (x eq y);
endtype
```

Fig. 13. 3-map coloring specification.

```
type 50-Queens is ConstraintProblem
      ...
      ofsort Nat
       (*Enter number of variables with varNum*)
       varNum = tens(5, 0); (*We have 50 queens*)
       (*Enter domain size of each variable with dSize()*)
       dSize(x) = tens(5, 0); (*Each queen can take one of the 50 positions*)
      ofsort Bool
       (*Define relations between two variables with rel()*)
       rel(x, x) = false; (*A queen cannot attack itself*)
       rel(x, y) = true;  (*Any two queens may attack each other*)
       (*Define satisfaction with sat()*)
       sat(x, m, y, n) = (m ne n) and abs(id(x)-id(y)) ne abs(m-n);
       (* Specifying constraint given in Fig.~1*)
       (*abs() for absolute value*)
      ...
endtype
```

Fig. 14. 50-Queens specification.

following is one possible solution: A, E and F have color red, B and D have color green, and C has color blue.

Let us consider the N-Queens problem defined in Example 1. As shown in Fig. 14, we instantiate the CSP template to produce the specification of, for instance, the 50-Queens. Note that the equations of *rel*() and *sat*() are defined for any number of queens. To change the number of queens, we just need to change the values of *varNum* and *dSize*().

4.2. *A Lotos template for TCSPs*

In order to handle temporal constraints, we extend the CSP template with two new operations: *dSOPO*() assigns a SOPO to an event, and *tRel*() defines which two events have a temporal relation between them. The template for describing a TCSP is illustrated in Fig. 15. The operations *dSize(), rel*() and *sat*() have default equations which are defined for any application under temporal constraints. The Lotos specification of a given temporal CSP is straightforward: the user only needs to enter the events, SOPO of each event and temporal relations between events.

It should be noted that the Allen algebra [25] for representing symbolic information has been extended here using Lotos equations in order to include numeric distances. For instance, we can use equations to specify the relation "$Event_1$ should finish within one hour after $Event_2$ finishes". This constraint cannot be handled by Allen primitives. We also note that in the data type Allen, the Allen primitives are defined as natural numbers in order to easily describe any disjunction of them.

We now show how to apply the TCSP template on the temporal reasoning problem given below.

Example 4 (A Temporal Reasoning Problem). John, Mike and Lisa work for the same company. It takes John 20 minutes, Mike 25 minutes and Lisa 30 minutes

```
type TemporalCSPTemplate is (*Uses*) ConstraintProblem, SOPO,
Allen
 sorts FixVar (*Set of domains*)
 opns (*Operations*)
      (*Enter all the events*)
      varNum: -> Nat (*Number of events*)
      dSize:FixVar -> Nat (*Domain size*)
      dSopo:FixVar -> SOPO (*SOPO*)
      id(*!external implementation*):FixVar -> Nat
      tRel:FixVar,FixVar -> Nat (*Relations*)
      (*Default operations*)
      rel:FixVar,FixVar -> Bool
      sat:FixVar,Nat,FixVar,Nat -> Bool
      \_eq\_,\_ne\_: FixVar,FixVar -> Bool
 eqns (*Equations*)
      forall x,y:FixVar, m,n:Nat
      ofsort Nat
       (*Enter number of events with varNum*)
       dSize(x) = getSize(dSopo(x)); (*Default equation*)
      ofsort SOPO
       (*Enter SOPO of each event with dSopo()*)
      ofsort Nat
       (*Define temporal relations between two events with tRel()*)
      ofsort Bool
       (*Default equations*)
       rel(x,y) = tRel(x,y) ne a\_all;
       (tRel(x, y) ne a\_all)=>
             sat(x,m,y,n)=tSat(getIvl(dSopo(x),m),getIvl(dSopo(y),n),tRel(x,y));
       (tRel(y,x) ne a\_all)=>
             sat(x,m,y,n)=tSat(getIvl(dSopo(y),n),getIvl(dSopo(x),m),tRel(y,x));
       sat(x,m,y,n) = true;
       x eq x = true; x eq y = false; x ne y = not (x eq y);
endtype
```

Fig. 15. A template for TCSPs.

to get to work. Every day, John leaves home between 7:20 and 7:26. Mike arrives at work between 7:55 and 8:00. Lisa arrives between 7:50 and 8:00. We also know that John and Mike meet at a traffic light on their way to work, Mike arrives to work before Lisa, and John and Lisa go to work at the same time.

We want to know whether this story is consistent (is possible to happen). This problem consists of both numeric and symbolic temporal information. As illustrated in Fig. 16, it can be transformed into a temporal CSP using TemPro model [16]. The constraints between the three events John, Lisa and Mike are the disjunction of *Allen* primitives, denoted by the symbol "|". For example, "John s|si|eq Lisa" states that John and Lisa go to work at the same time. The domain of each event is represented as a SOPO.

We now instantiate the TCSP template to produce the Lotos specification of this problem. In Fig. 17, we just show the parts entered by the user. The simulation

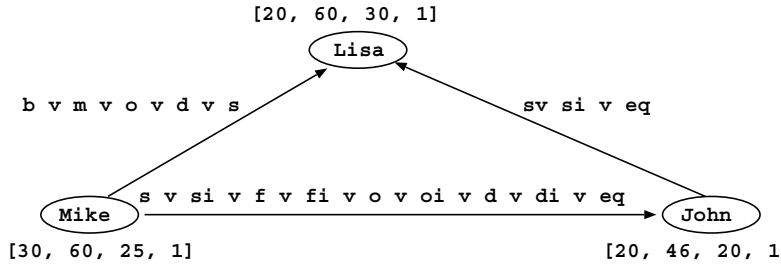

Fig. 16. Graph representation of temporal reasoning problem.

```
type TemporalReasoning is ConstraintProblem, SOPO, Allen
 sorts FixVar
 opns
     (*Enter all the events*)
      Mike, Lisa, John: -> FixVar
     ...
     (*Enter number of events with varNum*)
     varNum = 3; (*We have here three events*)
     (*Enter SOPO of each event with dSopo()*)
     dSopo(Mike) = sp(30, 60, 25, 1);
     dSopo(Lisa) = sp(20, 60, 30, 1);
     dSopo(John) = sp(20, 46, 20, 1);
     ...
     (*Define relations between two events with tRel()*)
     tRel(John, Mike) = a\_touches;
     tRel(Mike, Lisa) = a\_b+a\_m+a\_o+a\_d+a\_s;
     tRel(John, Lisa) = a\_s+a\_si+a\_eq;
     tRel(x, y) = a\_all;
     ...
endtype
```

Fig. 17. Temporal reasoning problem specification.

of this specification produces only one solution: {Mike = (30, 55), John = (26, 46), Lisa = (26, 56)}, where for instance (30, 55) is an interval of time starting at (7:)30 and ending at (7:)55.

5. Improving Lotos Simulation

In the previous section, we have seen how to specify in Lotos any constraint application using the generic CSP/TCSP template and its reusable library. The simulation of the specified CSP generates the solutions. However, the simulation is still very time consuming. For instance, the simulation of the 25-Queens takes more than 80 seconds. Therefore we propose here to integrate all the CSP algorithms introduced in Sec. 2 into the Lotos specifications in order to significantly increase the performance of the simulation. This integration is possible thanks to the external

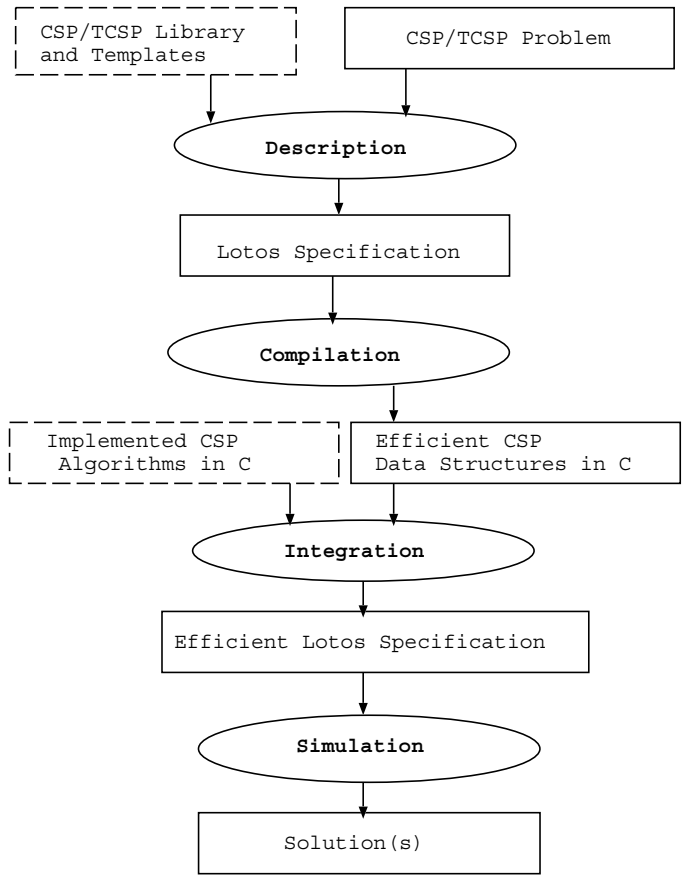

Fig. 18. Lotos-based (temporal) CSP framework.

implementation of data types. Indeed the CADP toolbox gives us the opportunity to use external operations for both sorts and operations of Lotos.

As shown in Fig. 18, first we automatically translate a CSP description into efficient data structures in C. Then we incorporate to these C types the CSP algorithms. In Fig. 19, we use and implement in C the five algorithms: Backtrack, AC-3, AC-3.1, Forward Checking (FC) and Full Look Ahead (FLA). The special comment (*!implementedby*) allows to use external implementation of the CSP algorithms. These algorithms can be used together for a better performance. For instance, AC-3 + FC (respectively AC-3 + FLA) means that arc consistency using AC-3 is performed first before applying FC (respectively FLA). For the purpose of implementation, we use bitmap for representing both domains and relations in order to improve time and space efficiency. In addition, because arc consistency algorithms [1] need to save and restore domains during backtrack, all the domain data are stored in a single memory space to expedite domains copying and restoring operations.

```
type Algorithms is opns
   BT (*!implementedby ALG\_BT*): -> Nat (*Backtrack*)
   AC3 (*!implementedby ALG\_AC3*): -> Nat (*AC-3*)
   AC31 (*!implementedby ALG\_AC31*): -> Nat (*AC-3.1*)
   FC (*!implementedby ALG\_FC*): -> Nat (*Forward Checking*)
   FLA (*!implementedby ALG\_FLA*):  -> Nat (*Full Look Ahead*)
eqns
   ofsort Nat
   BT = 1; AC3 = 2; AC31 = 4; FC = 8; FLA = tens(1, 6);
endtype
```

Fig. 19. Integration of CSP algorithms.

To get the solution (complete assignment of values to all variables that satisfies all the constraints) of a given (temporal) CSP using the implemented CSP algorithms, we just need to perform the operation *getSolution*() defined in a data type called Solution.

getSolution(CSP specification, CSP algorithms, number of solutions)

For example to get the two solutions of the 4-Queens problem using FC (as explained in Fig. 3), we execute the following statement: *getSolution(4-Queens, FC, 2)*. To return all the twelve solutions of the temporal reasoning problem of Example 4, we perform the following operation: *getSolution(TemporalReasoning, AC-3 + FLA, N)*. This action consists of simulating the specification TemporalReasoning (given in Fig. 17) using the two algorithms AC-3 and FLA.

Let us consider the scheduling problem of Example 2. A part of the specification of this problem is given in Fig. 20. To produce one solution to this problem, we execute the following operation: *getSolution(MachineScheduling, AC-3.1 + FC, 1)*.

6. Experimentation

In this section we present and discuss the results of the experiments we conducted in order to evaluate the time performance of our Lotos-based framework when dealing with various constraint problems. The tests are conducted on instances of the N-Queens problem presented in Example 1, the machine scheduling problem illustrated in Example 2 and randomly generated CSPs. Random CSPs are generated using the random uniform CSP generator described in [28]. All tests are performed on a 2.8 GHz Pentium IV computer under Linux and all procedures are coded in C language.

Figure 21 presents the results of the tests conducted on the machine scheduling problem. The goal here is to compare the time performance of the different techniques we described in Sec. 2. More precisely we evaluate the performance of the following methods.

AC-3+BT AC-3 is used in the preprocessing phase while BT is used during the search phase.

```
type MachineScheduling is ConstraintProblem, SOPO, Allen
 sorts FixVar
 opns
   (*Enter all the events*)
   AM1, AM2, AM3, BM1,..., EVT1: -> FixVar
    ...
   (*Enter number of events with varNum*)
   varNum = tens(1, 7);  (*We have 17 events*)
    ...
   (*Enter SOPO of each event with dSopo()*)
   dSopo(AM1) = sp(0, tens(2, 5), 3, 1);
   dSopo(AM2) = sp(0, tens(2, 5), 3, 1);
   dSopo(AM3)= sp(0, tens(2, 5), 6, 1);
   dSopo(BM1)= sp(0, tens(3, 0), 5, 1);
    ...
   (*Define temporal relations between two events*)
   tRel(AM2, AM3) = a\_b + a\_m;
   tRel(AM2, AM1) = a\_b + a\_m;
   tRel(AM1, AM3) = a\_b + a\_m + a\_bi + a\_mi;
    ...
endtype
```

Fig. 20. Specification of machine scheduling problem.

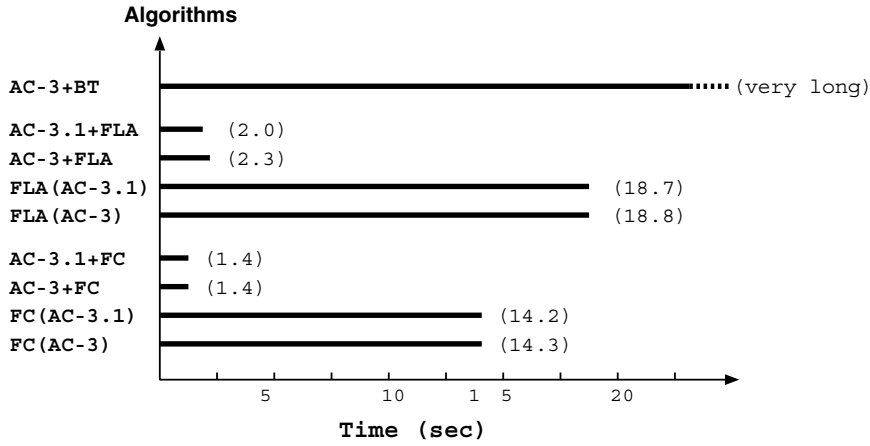

Fig. 21. Benchmarks of the scheduling problem.

AC-3.1+FLA AC-3.1 is used in the preprocessing phase while FLA (including AC-3.1) is used in the backtrack search phase.

AC-3+FLA AC-3 is used in the preprocessing phase while FLA (including AC-3) is used in the backtrack search phase.

FLA(AC-3.1) FLA using AC-3.1 is used in the backtrack search phase (there is no preprocessing phase here).

FLA(AC-3) FLA using AC-3 is used in the backtrack search phase.

AC-3.1+FC AC-3.1 is used in the preprocessing phase while FC (including AC-3.1) is used in the backtrack search phase.

AC-3+FC AC-3 is used in the preprocessing phase while FC (including AC-3) is used in the backtrack search phase.

FC(AC-3.1) FC using AC-3.1 is used in the backtrack search phase (there is no preprocessing phase here).

FC(AC-3) FC using AC-3 is used in the backtrack search phase.

BT is the slowest method. As we mentioned in Sec. 2, the problem with this technique is that the algorithm fails several times for the same reason. The late detection of inconsistencies dramatically affects the performance of this method. We can, for example, easily see from Fig. 2 that there are more consistency checks in the case of BT for solving the 4-Queens than FC in Fig. 3 and FLA in Fig. 4.

The second point we notice is that applying AC-3 (or AC-3.1) before the search is very helpfull. AC-3.1+FC is 10 times faster than FC alone. The same can be said for AC-3.1+FLA and FLA alone. As we mentioned in Sec. 2, when applied before the search, AC-3 reduces the size of the search space which will speed up later the time needed for the search. We notice however that there is no big difference in terms of running time when using AC-3 or AC-3.1. Indeed AC-3.1 is mainly helpful for problems where variables are defined on very large domains.

The last point we highlight is that FC is slightly better than FLA. Indeed while FLA reduces more inconsistent values than FC (as we can see when comparing Figs. 3 and 4 in the case of the 4-Queens), the time effort needed to detect and remove these inconsistencies by FLA is a little expensive and does not really compensate the overall running time to get a solution.

Figure 22 shows the results of tests conducted on instances of the N-Queens problem. Each instance is defined by the number of queens. Both BT and FC (including AC-3.1) are used here without the preprocessing phase. Indeed applying AC-3.1 (or AC-3) in the preprocessing phase does not remove any inconsistent value in the case of the N-Queens. To no surprise, FC is more efficient than BT for all instances of the problem.

Table 2 shows the results of tests conducted on instances of random CSPs. Each problem instance is defined by the number of variables N, their domain size D (number of values within each domain), the percentage of compatible pairs of values within each relation and the percentage of possible constraints.[i] Each test result corresponds to the running time in seconds needed to get a solution when running the AC-3.1 + FC (AC-3.1) method on a particular problem instance. In the table, the symbol "..." denotes that a solution cannot be obtained. As we can see, the hardest problems are those with 50% incompatible pairs (that we call middle-constrained problems). Indeed, under-constrained and over-constrained

[i]Since we are dealing with binary constraints, the total number of possible constraints is equal to $N(N-1)/2$ where N is the number of variables.

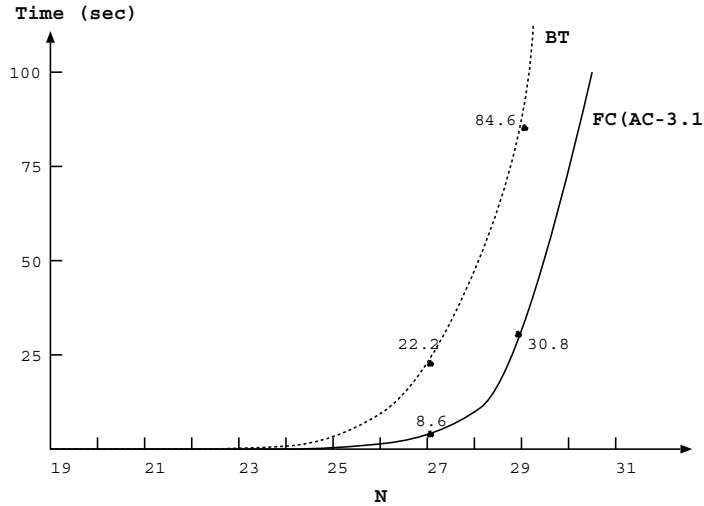

Fig. 22. Benchmarks of the N-queens problems.

Table 2. Benchmarks on random CSPs.

% of const	% of incomp pairs	Time (sec) N = 20, D = 20	Time (sec) N = 30, D = 30	Time (sec) N = 50, D = 50	Time(sec) N = 100, D = 100
100	20	0	0.2	0.7	12.1
100	**50**	0.6	1.8	32.5	...
100	80	0	0	0.2	8.3
75	20	0	0	0.6	7.8
75	**50**	0.5	12.3	144.3	...
75	80	0	0	0.3	11.4
50	20	0	0	0.8	45.12
50	**50**	2.2	7.8	27.3	125
50	80	0	0.3	0.3	30.9
25	20	0	0	0	2.1
25	**50**	0	0	17.12	28.1
25	80	0	0.2	0	0

problems (corresponding respectively to 20% and 80% incompatible pairs) are easy to solve. In the case of under-constrained problems, there are many solutions satisfying all the constraints and in the case of over-constrained problems the arc consistency at the preprocessing stage reduces considerably the size of the search space before the backtrack search.

7. Conclusion and Future Work

Today, there is a significant need to solve more complex combinatorial problems with more intelligent and time efficient solving techniques. A more challenging task

is the description of these problems which involves complex (temporal) constraints. The data part of Lotos provides a good means for specifying and solving these problems. We have significantly improved the simulation speed of Lotos through the CSP constraint propagation techniques. The integration of these algorithms is very useful to handle large combinatorial problems in practice since it can prevent the problem from becoming practically unsolvable too quickly.

In the future, we are interested in building a user friendly tool that assists in the construction of CSP specifications in an incremental manner. Considering that Lotos process part is better than the data part to describing dynamic actions, it is possible to use Lotos processes to specify the addition|retraction of constraints in a dynamic way. This is motivated by the fact that most real-world constraint problems are evolving in time and we have to deal in this case with any new information that corresponds to a constraint addition|relaxation. The other reason for handling dynamic constraints is to allow the user to interact with the solver in a more efficient way. After specifying a given constraint problem in Lotos, the user can, for example, add or remove some constraints and see the effect of this change in an incremental way.

Another perspective is to extend the solving techniques by including incomplete methods such as Stochastic Local Search (SLS) [29], Genetic Algorithms (GAs) [30], Ant Colony Algorithms (ACAs) [31] and Neural Networks [32]. While these techniques do not always guarantee a complete solution to the problem, they are very efficient in time (comparing to CSP solving methods) and can thus be useful if we want to trade the quality of the solution for the time performance.

References

1. R. Haralick and G. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14** (1980) 263–313.
2. V. Kumar, Algorithms for Constraint Satisfaction Problems: A survey, *AI Magazine* (1992).
3. A. K. Mackworth, Consistency in networks of relations, *Artificial Intelligence* **8** (1977) 99–118.
4. A. K. Mackworth and E. Freuder, The complexity of some polynomial network-consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25** (1985) 65–74.
5. U. Montanari, Fundamental properties and applications to picture processing, *Information Sciences* **7** (1974) 95–132.
6. T. Bolognesi and E. Brinksma, Introduction to the ISO specification language Lotos, *Computer Networks and ISDN Systems* **14** (1987) 25–59.
7. J. C. Fernandez, H. Garavel, A. Kerbrat, and L. Mounier, CADP: A protocol validation and verification toolbox, Lecture Notes in Computer Science 1102, 1996.
8. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic model checking for real-time systems, in *7th Symposium of Logics in Computer Science*, Santa-Cruz, California (IEEE Computer Science Press, 1992), pp. 394–406.
9. G. Holzmann, Algorithms for automated protocol validation, in *AT&T Technical Journal* **60** (1990) 32–44.

10. M. Bozga, J. C. Fernandez, and L. Ghirvu, Using static analysis to improve automatic test generation, in *Tools and Algorithms for Construction and Analysis of Systems*, 2000, pp. 235–250.
11. J. P. Krimm and L. Mounier, Compositional state space generation from Lotos programs, in Brinksma, E. (ed.), *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'97*), Enschede, The Netherlands (Springer-Verlag, 1997). Extended version with proofs available as Research Report VERIMAG RR97-01.
12. R. Dechter, *Constraint Processing* (Morgan Kaufmann, 2003).
13. I. Solver, http://www.ilog.com/products/solver/ (2007).
14. D. Diaz and P. Codognet, Design and implementation of the GNU prolog system, *Journal of Functional and Logic Programming* **6** (2001).
15. M. Mouhoub, F. Charpillet, and J. Haton, Experimental analysis of numeric and symbolic constraint satisfaction techniques for temporal reasoning, *Constraints: An International Journal* **2** (1998) 151–164.
16. M. Mouhoub, Reasoning with numeric and symbolic time information, *Artificial Intelligence Review* **21** (2004) 25–56.
17. R. Mohr and T. Henderson, Arc and path consistency revisited, *Artificial Intelligence* **28** (1986) 225–233.
18. P. van Hentenryck, Y. Deville, and C. M. Teng, A generic arc-consistency algorithm and its specializations, *Artificial Intelligence* **57** (1992) 291–321.
19. C. Bessière, Arc-consistency and arc-consistency again, *Artificial Intelligence* **65** (1994) 179–190.
20. C. Bessière, E. Freuder, and J. Regin, Using inference to reduce arc consistency computation, in *IJCAI'95*, Montréal, Canada, 1995, pp. 592–598.
21. R. J. Wallace, Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs, in *IJCAI'93*, Chambery, France, 1993, pp. 239–245.
22. Y. Zhang and R. H. C. Yap, Making ac-3 an optimal algorithm, in *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA, 2001, pp. 316–321.
23. C. Bessière and J. C. Régin, Refining the basic constraint propagation algorithm, in *Seventeenth International Joint Conference on Artificial Intelligence* (*IJCAI'01*), Seattle, WA, 2001, pp. 309–315.
24. M. Mouhoub, F. Charpillet, and J. P. Haton, Experimental analysis of number and symbolic constraint satisfaction techniques for temporal reasoning, *Constraints: An International Journal* **2** (1998) 151–164.
25. J. F. Allen, Maintaining knowledge about temporal intervals, *Communications of the ACM* **26** (1983) 832–843.
26. M. Mouhoub, S. Sadaoui, and A. Sukpan, Formal description techniques for CSPs and TCSPs, in *Proceedings of International Conference on Software Engineering & Knowledge Engineering* (*SEKE04*), 2004, pp. 406–410.
27. M. Mouhoub and S. Sadaoui, Improving lotos simulation using constraint propagation, in *Proc. of the 17th IEEE International Conference on Tools with Artificial Intelligence* (*ICTAI'05*), Hong Kong, November 14–16, 2005.
28. Random Uniform CSP Generators, http://www.lirmm.fr/∼bessiere/generator.html (2004).
29. B. Selman and H. A. Kautz, An empirical study of greedy local search for satisfiability testing, in *AAAI'93*, 1993, pp. 46–51.

30. B. Craenen and A. Eiben, Comparing evolutionary algorithms on binary constraint satisfaction problems, *IEEE Transactions on Evolutionary Computation* **7** (2003) 424–444.
31. T. Stützle and H. Hoos, Improvements on the ant system: Introducing the max-min ant system, *Artifcial Neural Networks and Genetic Algorithms* (1998) 245–249.
32. J. Hopfield and D. Tank, Neural computation of decisions in optimization problems, *Biological Cybernetics* **52** (1985) 141–152.